The Journal of Logic and Algebraic Programming 80 (2011) 3–12

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

On the potential advantages of exploiting behavioural information for contract-based service discovery and composition

Antonio Brogi

Department of Computer Science, University of Pisa, Italy

ARTICLE INFO

Article history:
Available online 2 March 2010

Keywords:
Service-oriented computing
Service descriptions
Behaviour information

ABSTRACT

The importance of service contracts providing a suitably synthetic description of software services is widely accepted. While different types of information – ranging from extra-functional properties to ontological annotations to behavioural descriptions – have been proposed to be included in service contracts, no widely accepted de facto standard has yet emerged for describing service contracts, except for signature information. The lack of a de facto standard is inhibiting large scale deployment of techniques and tools supporting enhanced discovery and composition of services.

In this paper we discuss the potentially huge advantages of exploiting behavioural information for service discovery and composition, and relate them to the cost of generating such information and to the needed trade-off between expressiveness and cost and value of analysing such information. On such ground, we also discuss the potential suitability of some well-known modelling approaches to become the de facto standard to represent service behaviour in contracts, also in view of contextual factors (such as required know-how and current employment).

1. Introduction

Service-oriented computing [30] builds on the very notion of (software) *service*. Simply stated, a software service is a software component that can be discovered and used on the net, via public domain standards, by other software components. Business processes play a crucial role in the integration of services both within and across enterprise boundaries. The push towards business process automation – strongly motivated by the enormous opportunities in terms of cost and quality of service provision – calls for effective solutions to the problem of integrating different enterprise applications involved in such processes. Service-oriented computing is considered today to be one of the most promising approaches to address the automation of business processes and, more in general, the integration of software systems. Clearly, the possibility of easily integrating software services has opened a new horizon in software development, introducing at the same time new technological challenges which mainly originate from the difficulties of guaranteeing the correct interoperation of remote services developed and provided by different parties.

One of the crucial aspects of services in general is that consumers use services while having little or no knowledge of how those services are implemented. Car repairing or air transportation are just two examples of services that are used every day by customers with little or no knowledge of how those services are implemented. Indeed customers typically choose to use a service on the basis of the *service contract* which is (more or less explicitly) exposed by the service provider. Informally, a service contract should describe all the terms of usage of the service, including cost, warranties, possible compensations, as well as all the other aspects that could possibly be object of controversy between the customer and the provider.

E-mail address: brogi@di.unipi.it

1567-8326/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2010.01.001

The role of service contracts is even more critical in the case of software services, where both service provision and usage are performed by automata. The need of including signature information in software service contracts to enable interoperability among services is universally accepted. WSDL (Web Service Description Language [46]) is the de facto standard to define the (syntax of the) functionalities featured by a Web service. While many proposals have been put forward to include other types of information in service contracts – ranging from extra-functional properties to ontological annotations to behavioural descriptions, no widely accepted de facto standard has yet emerged for describing service contracts, beyond signature information.

In this paper we will discuss the potentially huge advantages of exploiting behavioural information for service discovery and composition, and relate them to the cost of generating such information and to the needed trade-off between expressiveness and cost and value of analysing such information.

It is worth saying that while we will also mention some of the issues related to including ontology annotations in service contracts, we will not touch other important aspects of service contracts, such as the many proposals for including QoS information and the definition of new types of Service Level Agreements (capable of suitably formalising different aspects of service provision, such as access cost, warranties, or compensations), nor the orthogonal issues related to contract validation and monitoring.

2. What type of information should be included in service contracts?

As we already mentioned in Section 1, the need of including *signature information* in service contracts to enable interoperability among services is universally accepted. In the Web service arena, WSDL [46] has prominently emerged as the de facto standard for defining the (syntax of the) functionalities featured by services. WSDL permits to syntactically describe the operations offered by a Web service in terms of the type of messages that a service can send or receive. Service providers publish WSDL advertisements on UDDI [40] registries, which in turn provide clients with keyword- and taxonomy-based service discovery capabilities. Once a client knows the WSDL description of a service, it can send and receive SOAP [37] messages to interoperate with such service. The importance of *Quality of Service* (QoS) properties has also been recognised since the very birth of Web services (e.g., see [21,42]). QoS information covers different extra-functional aspects of service provision, ranging from *availability* (e.g., percentage of time a service is operating), to *performance* (e.g., latency, throughput), to *security* properties (e.g., confidentiality, message integrity, non-repudiation).

Many proposals have been put forward for including also other types of information in service contracts, besides signature and QoS information. In particular, a considerable amount of work has been devoted to propose the inclusion of ontology annotations and of behavioural descriptions in service contracts (Fig. 1). However, in spite of the important theoretical advances that have been achieved in these areas, the actual adoption of these extensions in the development world seems to be still quite far to come. In this paper we discuss the potential impact of those enriched descriptions and put them in perspective with the associated costs and potential benefits.

The choice of what type of information should be included in service contracts obviously depends on what contracts are to be used for. For instance, the purely syntactic nature of WSDL descriptions heavily limits the possibility of automating the service discovery task. Even a small syntactic difference in the syntax of operations or messages may spoil the interoperability between two services. Many authors have advocated the inclusion of *ontological annotations* to overcome non-relevant differences in the syntactic description of services. Some of the most known proposals in this sense are OWL-S [29], WSDL-S [2], METEOR-S [36], SWSO [34], or WSMO [47]. The use of ontologies for annotating data exchanged by software services is receiving also increasing attention in the e-health sector (e.g., see [38]), where ontological annotations offer an effective solution for overcoming the heterogeneity of health data representations adopted in different countries. Indeed, the use of ontologies to make the meaning of data explicit (e.g., with OWL [44]) currently seems to be spreading faster than the use of ontologies to provide semantic descriptions of services (e.g., with OWL-S). In any case, the very idea of employing ontologies to associate data with meaning calls for effective techniques to reason with different ontologies, as different people employ different ontologies in absence of universal standards. The most important obstacles to the spread of ontological annotations in software services do not seem to be of technological nature, however. On the one hand, the still scarce availability of friendly tools facilitating the annotations of data does not reduce the know-how required by software developers to enrich

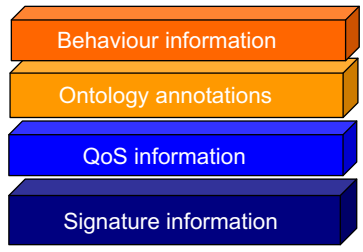

Fig. 1. Different types of information in a service contract.

their services with ontological annotations. On the other hand, and more importantly, a true spread of semantic services will only start when the derived advantages will become of clear interest for the market.

So far we have mentioned how signature information is needed in service contracts to enable interoperability, how QoS information can provide a valuable characterisation of extra-functional properties, and how ontological-annotations may permit to overcome syntactic differences in the format of data and service operations. Still, it is important to observe that the inclusion of all the above mentioned types of information in service contracts does not suffice to be able to guarantee a priori that service interaction will "really" work. This is due to the lack of information on the *interaction behaviour* of services. Indeed the order in which the operations of a service are to be invoked may lead to a dead-lock during the interaction with another service, if the interaction behaviour of the latter is not compliant with the interaction behaviour of the former. Unfortunately, WSDL interfaces provide syntactic descriptions of the operations offered by a service but not of the (partial) order in which those operations are offered by the service, and neither QoS nor ontological descriptions provide such a behavioural information.

It is important to stress that the absence of behavioural information in service contracts really compromises the possibility of guaranteeing a priori that a composition of services will actually work. Indeed one of the key principles of service orientation is that service providers publish only the contracts of their services, and not the details of the implementation of those services. Therefore, if service contracts do not include any behavioural information, no a priori analysis can be performed to guarantee the correct interoperation of a service composition.

In the remainder of the paper, we focus on discussing what type of behavioural information could be included in service contracts, in relation to the cost of generating such information and to the cost and value of analysing such information.

3. How much behavioural information should be included in service contracts?

The key choice to make in order to include behavioural information in service contracts is, intuitively speaking, "how much behaviour" to represent in a service contract. Clearly, the full details of the behaviour of a service cannot be described in a contract. No provider intends to disclose to the net the details of the services she offers, for obvious business and security reasons. Therefore, the service behaviour that can be exposed in a contract must necessarily be an abstraction of the concrete behaviour of such service.

As pointed out in [22], the exposed service behaviour should ideally synthesise the "essential" aspects by hiding all the "unnecessary" details of the concrete behaviour of a service and, at the same time, enable the verification of "interesting" properties of service compositions. To make the above statement a bit more formal, consider the concrete service behaviour b_1 and b_2 and their corresponding abstractions $\alpha(b_1)$ and $\alpha(b_2)$ illustrated in Fig. 2.

The abstraction function α should be chosen so as to ensure a direct correspondence between "interesting" properties of concrete composites (e.g., dead-lock freedom) and properties of the corresponding abstractions. Ideally, the abstraction function should permit to condition the validity of a property $P(b_1 \| b_2)$ of a concrete composite[1] to the validity of some property $\Pi(\alpha(b_1)|\alpha(b_2))$ of the corresponding abstraction, so as to permit to verify on the abstractions properties of the concrete behaviour. In other words, given a property P over concrete behaviour, ideally the chosen abstraction function α should permit to determine a property Π over abstract behaviour such that the implication

$$\Pi(\alpha(b)) \Rightarrow P(b)$$

holds for any concrete behaviour b.

On the other hand, as higher abstractions may reduce the cost of analysis of abstract behaviour, abstraction functions that loose much concrete information become of practical interest. This leads to interpreting properties of abstract composites only as necessary conditions for the validity of the corresponding properties of concrete compositions. In other words, whenever property $\Pi(\alpha(b_1)|\alpha(b_2))$ can be disproved at the abstract level, we can conclude that the corresponding property $P(b_1 \| b_2)$ will not hold at the concrete level. Formally, given a property P over concrete behaviour, the chosen abstraction function α should permit to determine a property Π over abstract behaviour such that the implication

$$P(b) \Rightarrow \Pi(\alpha(b))$$

that is

$$\neg\Pi(\alpha(b)) \Rightarrow \neg P(b)$$

holds for any concrete behaviour b. In this sense, intuitively speaking, reasoning on abstractions of behaviour somehow resembles type-checking in conventional programming languages, where the occurrence of run-time errors in the execution of well-typed programs is not (fully) prevented. The value of the verification performed on abstract behaviour should hence be interpreted in terms of the amount of undesired concrete behaviour that will pass the abstract verification, as graphically illustrated in Fig. 3.

[1] To account for the syntactic differences between the concrete and abstract languages, in Fig. 2 we use a different symbol ("|") to represent in the abstract language the composition operator ("‖") employed in the concrete language.

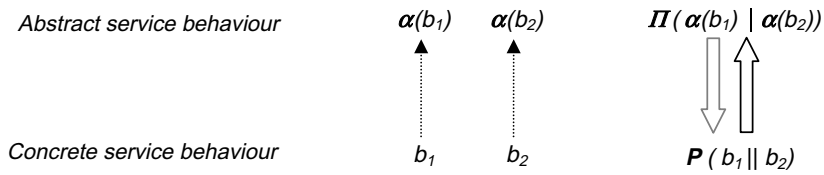

Fig. 2. Reasoning on abstract behaviour of services.

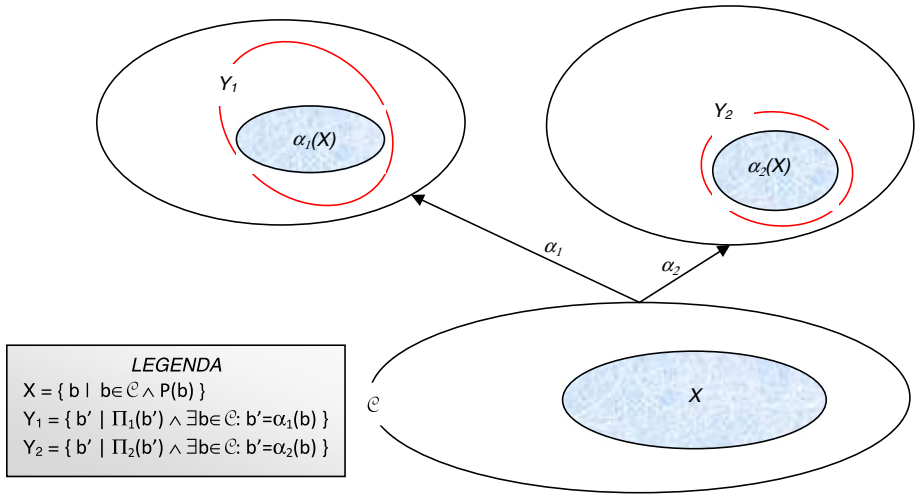

Fig. 3. Set-theoretic view of the implication relation between a property P of the concrete behaviour and two properties Π_1 and Π_2 over different behavioural abstractions.

In other words, the value of the verification performed by checking a property Π_1 over the set of abstract behaviour $\{b'|\exists b \in \mathscr{C} : b' = \alpha_1(b)\}$ can be measured in terms of the amount of concrete behaviour that do not satisfy the desired property P and that still however pass the abstract verification, that is in terms of the set

$$\begin{aligned} Y_1 \backslash \alpha_1(X) &= \{b'|\Pi_1(b') \wedge \exists b \in \mathscr{C} : b' = \alpha_1(b)\} \backslash \{b'|\Pi_1(b') \wedge \exists b \in \mathscr{C} : b' = \alpha_1(b) \wedge P(b)\} \\ &= \{b'|\Pi_1(b') \wedge \exists b \in \mathscr{C} : b' = \alpha_1(b) \wedge \neg P(b)\} \end{aligned}$$

Conceptually, the abstraction function over the concrete behaviour is inversely proportional to the cost and value of the analysis of abstract behaviour. Namely, while the analysis of concrete behaviour obviously brings the highest cost and value, the higher the abstraction the lower the cost and value of the analyses of the obtained abstract behaviour. The choice of what behavioural information to include in service contracts must hence be a trade-off between expressiveness and cost and value of the analyses. In the next section, we try to put in this perspective some of the approaches that have been proposed for modelling the behaviour of software services.

4. Some candidates for expressing service behaviour in contracts

In this section we discuss the potential suitability for service contracts of three well known approaches to modelling the behaviour of software systems: finite state machines, behavioural types, and workflows.

4.1. Finite state machines

Finite state machines have been long – and are still widely – employed to model the interaction behaviour of software systems [43]. The conceptual simplicity of finite state machines has enormously contributed to their widespread adoption for describing interactive systems. Their simplicity also enables efficient automated analyses of properties like compatibility or replaceability of protocols (expressed via finite state machines). On the other hand, the same simplicity of finite state machines imposes a severe bound on the expressiveness of the formalism, and the specification of an interaction protocol by means of a finite state machine necessarily abstracts away a considerable part of the complexity of the interaction.

We will discuss next the suitability of finite state machines to describe behaviour in service contracts by briefly reviewing three examples of approaches that have been proposed to model the interaction behaviour of services. (Note that the objective

of the following discussion is only to highlight some of the advantages and disadvantages that are common to finite state machine approaches – the discussion does not absolutely intend to represent an exhaustive survey of those approaches.)

One of the first proposals for expressing the interaction behaviour of object-oriented systems with finite state automata is presented by Nierstrasz in [25], where the notion of *regular type* is introduced to characterise the non-uniform behaviour of services. Simply stated, regular types are non-deterministic finite state automata whose transitions are labelled by names of methods and their associated parameters. The finiteness of the representation is achieved by abstracting method parameters into data types, and by consequently turning data-flow dependent choices into non-deterministic choices. As one may expect, the simplicity of the formalism permits an efficient verification of properties like the compatibility or the replaceability of regular types. On the other hand, the limited expressiveness of regular types induces a considerable abstraction on the concrete behaviour of services. As a consequence, proving for instance the compatibility of the regular types of two services provides a quite limited guarantee that the interaction of the concrete processes will not dead-lock. A further expressiveness limitation of regular types is that they do not permit to model dynamic changes occurring during the execution of processes, such as the creation of new processes or of new communication channels.

A conceptually similar approach is described by Yellin and Strom in [49], where the use of *finite state grammars* to specify the protocols of software components is proposed. The finite state grammars of [49] extend the regular types of [25] by permitting to separate the invocation of operations from the reception of results, and by formalising a synchronous semantics of inter-process communication. As in [25], the simplicity of the formalism permits an efficient verification of the compatibility and of the replaceability of protocols expressed by finite state grammars. And, as in [25], the limited expressiveness of the formalism (no state parameters, no dynamic process/channel creation) introduces a considerable abstraction in the process of modelling a concrete service behaviour by means of a finite state grammar. Therefore, as in [25], properties established on the abstract descriptions cannot always provide strong guarantees that the corresponding properties on the concrete processes will hold.

The use of finite state machines to model the interaction behaviour of software components is advocated by de Alfaro and Henzinger in [14] too, where the possibility of labelling state transitions also with internal actions is introduced. A weaker "optimistic" notion of compatibility is considered (namely, two automata are considered compatible if there is at least one environment in which they will be able to interoperate successfully) in order to determine environmental conditions ensuring the interoperability of two automata. As for [25,49], the simplicity of the approach enables the efficient verification of automata properties, such as (weak) compatibility and a notion of behavioural refinement. Still, the approach described in [14] maintains the same limitations (no state parameters, no process/channel creation) of the other approaches.

It is worth adding that, apart from the above mentioned expressiveness limitations, finite state machines have not really been explicitly proposed as true "behavioural types" to model the behaviour of software systems. In other words, little attention has been devoted so far to define *type systems* capable of converting a concrete behaviour of a service (written in some concrete programming language) into the corresponding abstract behaviour. This is instead one the very objectives of the proposals aimed at defining so called "behavioural types", as we will see in the next paragraph.

4.2. Behavioural types

The term "behavioural type" was first introduced by Meredith and Bjorg in [22], where they advocate the introduction of rigorous typing disciplines capable of synthesising the essential aspects of the interaction behaviour of services while at the same time enabling the verification of interesting properties of the behaviour of the typed systems.

The first example of behavioural types are the so called *session types* introduced by Honda, Vasconcelos and Kubo in [15]. The key idea of [15] is to provide a rigorous type system centered on the notion of *sessions*, intended as sequences of communication actions between two partners. A process algebra term (specifying a service behaviour) is typed into a set of sessions, that synthesise all possible interactions of the service with other parties. A notion of duality of (session) types is introduced in [50] to define a notion of type compatibility that guarantees (under certain conditions) the successful interoperation of (a pair of) source processes. An interesting aspect of session types is that, while data values are still abstracted into data types, session types permit to model session passing, thus partly accounting for the dynamic reconfiguration of the processes' topology. It is worth mentioning that the actual usability of session types for adapting services presenting mismatching behaviour has been demonstrated for instance in [6].

One of the most relevant examples of behavioural types are the so called *process types* introduced by Igarashi and Kobayashi in [16]. The typing rules presented in [16] permit to abstract a pi-calculus [24] term (that specifies a service behaviour) into a CCS [23] term, thus succeeding in the non-trivial task of typing (name) mobility. Process types can be exploited to prove safety properties (such as absence of dead-locks), as well as liveness properties (such as absence of live-locks) if actions are tagged with event names [17]. An important result of [16] is the definition of a general framework in which different behavioural type systems can be expressed by simply instantiating a subtyping and a consistency relation. Notably, the framework in [16] also permits to establish several properties – such as lock-freedom and race conditions- of the source (Pi) processes by checking the consistency of the corresponding (CCS) types.

One of the limitations of [16] is that type checking cannot be performed modularly, and this would hence inhibit the possibility of employing process types for service contracts. Chaki et al. enhanced process types in [13] precisely to enable the modular generation of process types by defining a type-and-effect system to translate Pi processes into CCS types by

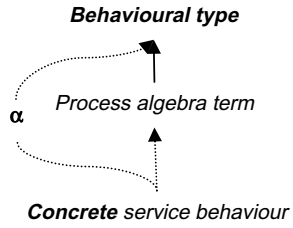

Fig. 4. The double abstraction introduced by behavioural types.

employing weak open simulation as the abstraction function. A prototype implementation of the type system of [13] can be used to experiment the methodology on CCS types with a finite number of states.

Rather than surveying here the body of significant theoretical results on behavioural types that have been achieved in the last few years [5], it is worth highlighting some of the factors that will delay – if not inhibit altogether – the adoption of behavioural types for describing behaviour in service contracts. One of them is obviously the availability of suitable *type-inference tools* to automatically generate the behavioural types of concrete services. Without such tools, service developers would be forced to hand-write the behavioural types of their services. Besides being a tedious and error-prone task, this would require service developers to own a know-how of the non-trivial details of behavioural types, and this will inhibit the adoption of this technology. It is also worth recalling here that type-inference tools – and hence the underlying theory of behavioural types – should also feature the possibility of modularly typing parts of multi-part interaction protocols that emerge from the interaction of separate services. For the same reasons of needed know-how and ease of use, the potential spread of behavioural types will require the availability of *type-checking tools* to automatically verify properties over behavioural types. In this perspective, a further important feature of type-checking tools – and hence of the underlying theory of behavioural types – is whether, and how easily, they will allow users to state and verify non-predefined properties. While the efficiency of type-inference tools may not be an issue (as service contracts are to be generated off-line), the efficiency of type-checking tools will determine whether they could be eventually integrated in the service discovery process or rather employed only during the analysis phase. Last, but not least, most proposals of behavioural types consider process algebras as the source languages to be typed. This implies that – to avoid requiring service developers to be acquainted with process algebras – type-inference tools should be suitably extended or integrated so as to be able to generate behavioural types starting from the concrete behaviour of services, as sketched in Fig. 4.

Unfortunately, the great potential of behavioural types for modelling and verifying abstract behaviour has not been accompanied yet by the availability of engineered tools that could enable the penetration of this technology in the development world. Meanwhile, the use of workflows at different stages of service development has gained momentum in recent years, as we are going to discuss in the next paragraph.

4.3. Workflows

Traditionally workflows have been widely used in business process modelling, and workflow languages have become one of the pillars of software services, thanks in particular to the spread of WSBPEL, the Web Services Business Process Execution Language [45], which has been recently adopted as the OASIS standard for orchestrating software services.[2]

WSBPEL is an executable language that permits to specify a business process as a suitable composition of activities, some of which can be performed by other services exposed as WSDL services by business partners. A WSBPEL process is itself exposed as a WSDL service, that can hence in turn be invoked by other services. WSBPEL features a set of basic activities (e.g., for sending and receiving messages, or for signalling faults) and a set of structured activities that supports standard sequential, parallel, conditional, iterative and (non-deterministic) choice composition operators, as well as a non-trivial handling of faults, events and compensations.

One of the reasons of the success of WSBPEL is the two-level programming model it offers, that permits a clear and valuable separation of concerns between business model experts and programming experts. On one hand, WSBPEL features a *programming-in-the-large* level, where business model experts can specify the intended business workflow without having to be programmers. On the other hand, programming experts are responsible to fill all the *programming-in-the-small* details needed to convert an abstract business process into an executable process.

It is worth noting that, besides its use during the specification phase, the descriptive role of abstract WSBPEL processes can be also exploited to define the observable behaviour of the service implementing the business process. However, the intended semantics of non-trivial WSBPEL processes is not always easy to understand, mainly because of the presence of synchronisation links (introducing an arbitrary number of dependencies among activities) and of the arbitrarily deep nesting of activities, especially of those handling faults, events and compensations. The complexity of the semantics of WSBPEL has motivated the development of transformational semantics supporting the translation of BPEL processes into Petri nets [41,28]

[2] A discussion on how BPMN diagrams [26] can be transformed into WSBPEL processes can be found for instance in [27,33].

or workflows [9] in order to get explicit, easy to understand representations of BPEL processes. Various characterizations of business processes in terms of process algebras have been proposed too. For instance several soundness properties are formally characterised in [32] as invariants of Pi-calculus mappings of business processes.

The impressive spread of WSBPEL, and the multiple uses of WSBPEL abstract processes, are hence naturally pushing the possibility of directly employing workflows as the formalism to describe service behaviour in contracts. One of the notable advantages of choosing workflows to represent the abstract behaviour of services is that they can be formally analysed by means of the wealth of tools available for workflows or Petri nets (e.g. [48,31]) that feature the possibility of executing fully automated verifications as well as of analysing service behaviour by means of graphical interfaces providing a straightforward representation of service behaviour.

It is also worth noting that the control-flow and the data-flow of a service behaviour can be easily distinguished in a workflow, and this possibility notably simplifies the implementation of different types of analyses over the same representation of behaviour. The actual usability of workflow representations of WSBPEL processes for composing and adapting the behaviour of services has been already demonstrated for instance in [10,11,8].

Summing up, while workflows can be used to represent both the control-flow and the data-flow of a service behavior, behavioural types abstract from the data flow, and finite state machines further abstract from the dynamic creation of processes or channels. In this sense, we can say that the behavioural abstraction provided by finite state automata is higher than that provided by behavioural types, which is in turn higher than that provided by workflows. Conversely, because of the amount of information to be analysed, the cost and value of analysis tends to decrease from workflows to behavioural types to finite state automata. On the other hand, other important factors will contribute to determine which formalism may become the reference for expressing service behaviour. In such a perspective, we argue that the relative simplicity of the formalism, its widespread use for different purposes in the development process, the availability of engineered tools for simulation and analysis, all seem to suggest the possible emergence of workflows as a de facto reference for describing service behaviour.

5. Concluding remarks

We have started our discussion from the observation that the lack of behavioural information in service contracts inhibits the possibility of guaranteeing that service interactions will really work. This is due to the fact that the order in which the operations of services are to be invoked can lead to locks in the services' interaction (e.g., see [10,8]). According to one of the key principles of service orientation, providers publish only the contracts of their services, and hence if no behavioural information is included in service contracts, no a priori analysis can be performed to guarantee the correct interoperation of service compositions. We have then discussed "how much" behavioural information could be included in a service contract, in particular with reference to the need of a trade-off between expressiveness, and cost and value of analysing such information. In this perspective, we have then discussed the potential suitability for service contracts of three well-known approaches to modelling software behaviour: finite state machines, behavioural types, and workflows. We have also mentioned the importance of other factors that will contribute to determining whether or not behavioural information will be eventually included in service contracts, and which formalism will be chosen for that end. Those factors include the amount of know-how required by developers for handling behavioural information, the availability of engineered tools supporting the generation and the analysis of behavioural information, and of course the economical advantages for service providers of investing in the management of behavioural information.

At the end of Section 4, we have argued that the impressive spread of WSBPEL is suggesting the possible emergence of workflows as a de facto reference for describing service behaviour. It is worth stressing here that while such a possible trend could push the adoption of workflows to represent service behaviour in contracts, it would not per se require all analysis and verification tools to work necessarily on workflows. The plethora of techniques developed for Petri nets, behavioural types or finite state machines could of course be exploited to analyse abstractions of the workflows, obtained by means of suitable translators as sketched in Fig. 5. However, as already discussed in Section 3, to avoid requiring developers to own specific know-how of those techniques, the whole conversion and analysis process should be made transparent to developers by suitable engineered tools.

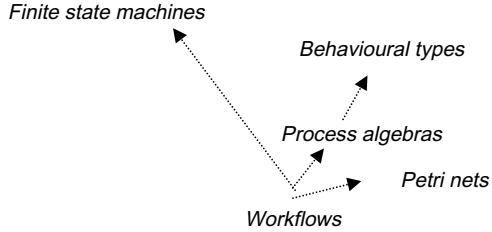

Fig. 5. Mapping workflows into other formalisms.

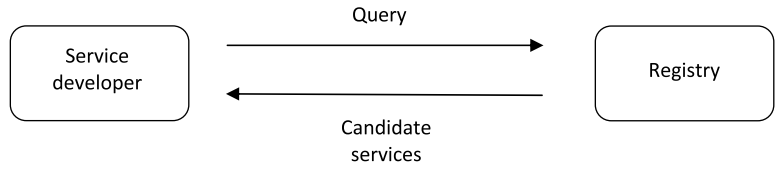

Fig. 6. Querying a service registry.

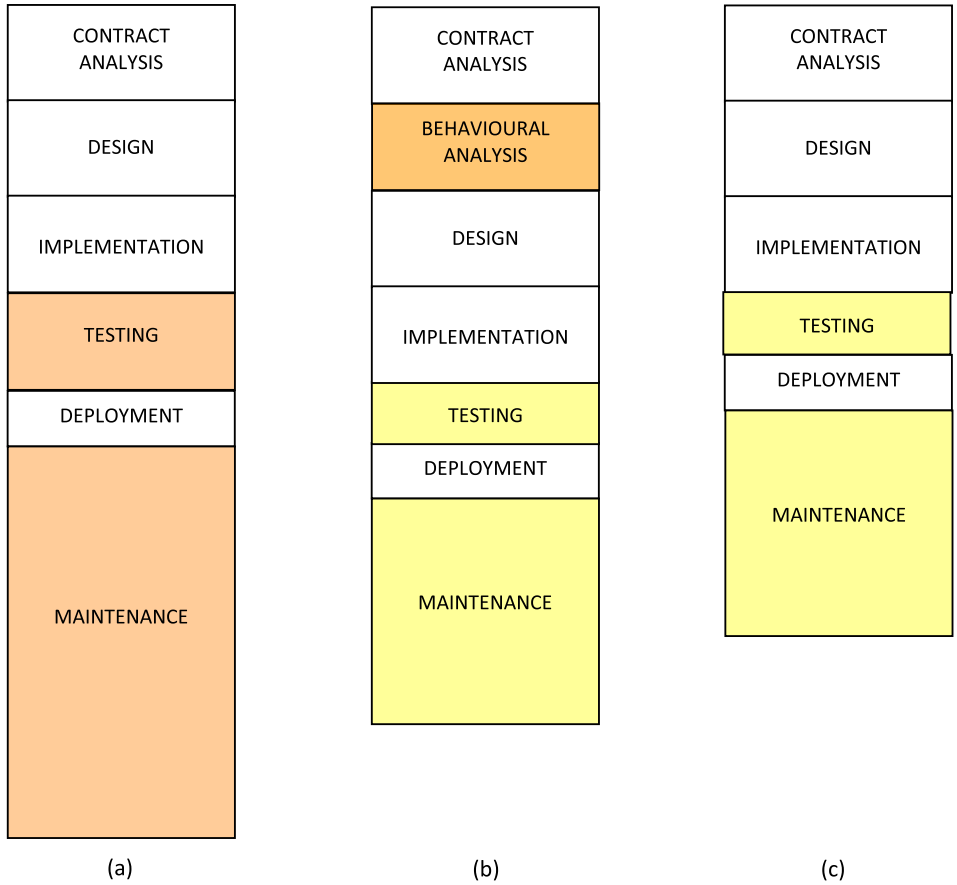

Fig. 7. Indicative illustration of the potential impact of behavioural analysis on the labour cost needed to provide services. The three situations are: (a) no behavioural analysis, (b) manual behavioural analysis, and (c) automated behavioural analysis.

In the remainder of this section, we will try to highlight some of the potentially huge advantages of exploiting behavioural information in the development of software services. Consider the typical situation of a service developer wishing to implement a new service by suitably orchestrating two available types of services. As a first step, the developer needs to locate two actual services that provide the desired functionalities. To do this, the developer may query one of the service registries available on the net, as illustrated in Fig. 6.

By issuing two queries (one for each type of service needed), the developer will fetch two sets of contracts of candidate services, that is, of services whose contracts match the specified requirements. At this point, the developer can analyse the contracts received in order to choose one service per set and start the design of the orchestrated service. As we already mentioned, without any information on the interaction behaviour of the chosen services, the developer cannot formally guarantee a priori that the designed orchestration will not lock. The common practice is to perform a testing phase on the implemented prototype, and to face during maintenance all problems that will possibly occur after deployment.

If instead contracts would include a formal description of service behaviour, a rigorous behavioural analysis could be performed before implementing and deploying the orchestration service. While the introduction of such an analysis phase would increase the labour needed to reach the implementation phase, such an additional cost would be largely compensated by a reduced need of testing and by a considerable reduction of maintenance costs, as illustrated in parts (a) and (b) of Fig. 7. The reason is that, as we have discussed in Section 4, a rigorous behavioural analysis can permit to verify a priori safety

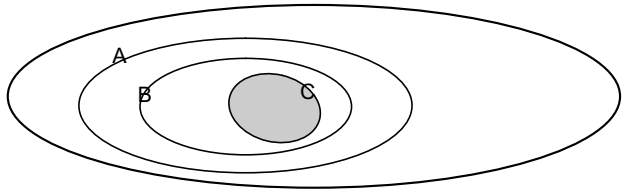

Fig. 8. Two-phase behavioural matching. The set of contracts matching the client query (A) is filtered during the discovery process (B), and then further filtered during the subsequent analysis phase (C).

properties of the composed services (such as lock-freedom), thus preventing – at design time – a considerable number of problems that would otherwise occur without such a verification phase. The potential advantages of introducing a behavioural analysis phase are even more evident if we consider the availability of engineered tools automating the required behavioural analysis. In such a scenario, the reduction of testing and maintenance costs would not be accompanied by additional labour cost for the (automated) behavioural analysis, as illustrated in part (c) of Fig. 7. It is worth stressing that the hypothesised availability of engineered tools featuring behaviour analyses – considered in the last scenario – is not unrealistic. Indeed, the considerable body of basic research on the behaviour of software systems has set firm theoretical grounds that make it already possible the development of effective analysis tools, as we briefly discussed in Section 4.

To overcome the current limitations of currently available service registries, which do not take into account behavioural information, various behaviour-aware matchmaking prototypes for service discovery have been proposed over the last few years (e.g. [7,4,39,18]). A very interesting feature of some of these prototypes is the possibility of handling *behavioural queries*. Simply stated, clients of registries may specify in the query the desired behaviour of the services they are looking for. The possibility of issuing behavioural queries turns out to be extremely valuable for instance when in need of substituting (either for technical reasons or for policy reasons due to changes in business partnerships) one or more services that are part of an orchestration. Indeed, the possibility of issuing a behavioural query specifying the interactive behaviour of the service to be replaced, and of discovering this way a new service featuring such a behaviour may dramatically reduce the cost of service substitution. Some of the proposed behaviour-aware matchmaking prototypes feature non-trivial behavioural analyses (e.g., bisimulation equivalence over Petri nets [3,12,19]) to implement strong matchings based on behavioural congruences, that are particularly valuable for modular service composition as they guarantee that the returned services can replace the old service without altering the behaviour of the overall orchestration. Also the notion of *operating guideline* of a service S, introduced in [20] to efficiently characterize the set of services that are behaviourally compatible with S (that is, whose composition with S is dead-lock free), can be directly applied to evaluate behavioural queries specifying the interactive behaviour of the service to be replaced as well as to verify service replaceability.

The main problem in implementing behaviour-aware analysis is, however, efficiency. Even considering the possibility of embedding behaviour-aware engines in UDDI registries – as in the approach taken by Srinivasan et al. in [35] to feature ontology-aware service matchmaking – the burden of performing non-trivial behavioural analyses spoils the scalability of service discovery with respect to the number of contracts stored in the registry. It is worth observing that no suitable indexing technique for behavioural descriptions has been proposed yet to face this scalability issue. While indexing is commonly employed to achieve scalable query-answering systems – Web search engines are the best known example – the definition of indexing over complex terms representing service behaviour is still an open research issue.

A suitable trade-off between accuracy and efficiency is hence needed in order to introduce behaviour-aware mechanisms in the service discovery process in a sustainable manner – that is, without degrading the latter to the point of spoiling its actual usability. In this perspective a possible approach to the problem may be to add on the matcher side suitable light-weight behaviour-aware mechanisms capable of efficiently performing simple checks on quite abstract behaviour (e.g., via finite state automata or even finite execution traces), and to delegate possible further, more sophisticated analyses to the client side. The potential value of such an early light-weight behavioural analysis can be measured in terms of the ratio between the introduced overhead and the percentage of contracts filtered out by such analysis (viz., $(|A| - |B|)/|A|$ in Fig. 8).

We have tried to highlight several important advantages of including behavioural information in service contracts, and we have tried to highlight some of the available results and some of the needed trade-offs that can make such an effort feasible. We insist however that several contextual factors (shared know-how, availability of engineered tools, investment/profit ratio) will play a crucial role in determining whether behavioural information will eventually enter as a first-class citizen in the service-oriented world.

References

[2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.T. Schmidt, A. Sheth, K. Verma, Web Service Semantics – WSDL-S Version 1.0, 2005. <http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.html>
[3] F. Bonchi, A. Brogi, S. Corfini, F. Gadducci, On the use of behavioural equivalences for Web services' development, Fund. Inform., 89 (4) (2009) 479–510.
[4] L. Bordeaux, G. Salaun, D. Berardi, M. Mecella, When are TwoWeb Services Compatible? Technologies for E-Services, LNCS, vol. 3324, Springer, 2005.

[5] A. Brogi, C. Canal, E. Pimentel, Behavioural types for service integration: achievements and challenges, Electron. Notes Theor. Comput. Sci. 180 (2) (2007) 41–54.
[6] A. Brogi, C. Canal, E. Pimentel, Behavioural Types and Component Adaptation, AMAST 2004, LNCS 3116, 2004.
[7] A. Brogi, S. Corfini, Ontology- and behaviour-aware discovery of service compositions, Int. J. Cooper. Inform. Syst. 17 (3) (2008) 319–347.
[8] A. Brogi, R. Popescu, Automated generation of BPEL adapters, in: Dan Lamersdorf (Ed.), ICSOC'06, LNCS, vol. 4294, Springer, 2006, pp. 27–39.
[9] A. Brogi, R. Popescu, From BPEL processes to YAWL workflows, in: 3rd International Workshop on Web Services and Formal Methods (WS-FM'06), LNCS, vol. 4184, 2006, pp. 107–122.
[10] A. Brogi, R. Popescu, M. Tanca, Design and implementation of SATOR: a web service aggregator, ACM Trans. Softw. Eng. Methodol. 19 (3) (2010).
[11] A. Brogi, R. Popescu, Service adaptation through trace inspection, Int. J. Business Process Integr. Manage. 2 (1) (2007) 9–16.
[12] G. Castagna, N. Gesbert, L. Padovani, A Theory of Contracts for Web Services, PLAN-X, Programming Language Technologies for XML, 2007.
[13] S. Chaki, S.K. Rajamani, J. Rehof, Types as models: model checking message passing programs, in: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2002, pp. 45–57.
[14] L. de Alfaro, T. Henzinger, Interface automata, Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109–122.
[15] K. Honda, V.T. Vasconcelos, M. Kubo, Language primitives and type disciplines for structured communication-based programming, in: ESOP'98, LNCS, vol. 1381, 1998, pp. 122–138.
[16] A. Igarashi, N. Kobayashi, A generic type system for the Pi-calculus, Theoret. Comput. Sci. 311 (1–2) (2004) 121–163.
[17] N. Kobayashi, A type system for lock-free processes, Inform. and Comput. 177 (2) (2002) 122–159.
[18] N. Lohmann, P. Massuthe, C. Stahl, D. Weinberg, in: S. Dustdar, J.L. Fiadeiro, A. Sheth (Eds.), Analyzing Interacting BPEL Processes, Business Process Management, LNCS, vol. 4102, Springer, 2006.
[19] A. Martens, Analyzing web service based business processes, in: M. Cerioli (Ed.), Fundamental Approaches to Software Engineering, LNCS, vol. 3442, Springer, 2005.
[20] P. Massuthe, Operating Guidelines for Services, Dissertation, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II; Eindhoven University of Technology, April 2009, ISBN: 978-90-386-1702-2.
[21] D.A. Menascé, QoS issues in web services, IEEE Internet Comput. 6 (6) (2002) 72–75.
[22] L.G. Meredith, S. Bjorg, Contracts and types, Commun. ACM 46 (10) (2003).
[23] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[24] R. Milner, Communicating and Mobile Systems: The Pi-calculus, Cambridge University Press, 1999.
[25] O. Nierstrasz, Regular Types for Active Objects, Object-Oriented Software Composition, Prentice-Hall, 1995, pp. 99–121.
[26] Object Management Group. Business Process Management Notation (BPMN), Version 1.2. <http://www.omg.org/docs/formal/09-01-03.pdf>
[27] C. Ouyang, M. Dumas, A.H.M. ter Hofstede, W.M.P. van der Aalst, From BPMN process models to BPEL Web services, Proceedings of 2006 IEEE International Conference on Web Services (ICWS'06), IEEE Press, 2006.
[28] C. Ouyang, E. Verbeek, W.M. van der Aalst, S. Breutel, M. Dumas, A.H. Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Sci. Comput. Program. 67 (2–3) (2007) 162–198.
[29] OWL-S: Semantic Markup for Web Services Version 1.1. <http://www.daml.org/services/owl-s/1.1/>
[30] M.P. Papazoglou, D. Georgakopoulos, Service-oriented computing, Commun. ACM 46 (10) (2003) 24–28.
[31] Petri nets World. <http://www.informatik.uni-hamburg.de/TGI/PetriNets/>
[32] F. Puhlmann, Soundness verification of business processes specified in the pi-calculus, in: R. Meersman, Z. Tari et al(Eds.)., OTM 2007, Part I, LNCS, vol. 4803, Springer-Verlag, Vilamoura, Portugal, 2007.
[33] J.C. Recker, J. Mendling, On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages, in: 18th International Conference on Advanced Information Systems Engineering, 2006.
[34] Semantic Web Services Ontology (SWSO) Version 1.0, 2005. <http://www.daml.org/services/swsf/1.0/swso/>
[35] N. Srinivasan, M. Paolucci, K.P. Sycara, An efficient algorithm for OWL-S based semantic search in UDDI, in: SWSWPC, 2004, pp. 96–110.
[36] A. Sheth, J. Miller, I.B. Arpinar, K. Verma, K. Gomadam, METEOR-S: semantic web services and processes, 2005. <http://lsdis.cs.uga.edu/projects/meteor-s/>
[37] Simple Object Access Protocol (SOAP). <http://www.w3.org/TR/soap/>
[38] The eHEALTH ontology project. <http://www.ehealthserver.com/ontology/>
[39] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, ASTRO: supporting composition and execution of web services, in: Proceedings of the Third International Conference on Service-Oriented Computing, LNCS, vol. 3826, 2005, pp. 495–501.
[40] Universal Description, Discovery and Integration (UDDI). <http://uddi.xml.org/>
[41] W.M.P. van der Aalst, H. Verbeek, Analyzing BPEL processes using Petri nets, in: D. Marinescu (Ed.), Proceedings of the 2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, 2005.
[42] W3C Working Group, QoS for Web Services: Requirements and Possible Approaches, 2003. <http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/>
[43] F. Wagner, Modeling Software with Finite State Machines: A Practical Approach, Auerbach Publications, 2006.
[44] Web Ontoloy Language (OWL). <http://www.w3.org/2004/OWL/>
[45] Web Services Business Process Execution Language (WSBPEL). <www.oasis-open.org/committees/wsbpel/>
[46] Web Services Description Language (WSDL). <http://www.w3.org/2002/ws/desc/>
[47] Web Service Modeling Ontology (WSMO) D2v1.2, 2005. <http://www.wsmo.org/TR/d2/v1.2/>
[48] M.T. Wynn, H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. ter Hofstede, D. Edmond, Business process verification – finally a reality! Business Process Manage. J. 15(1) (2009) 74–92.
[49] D.M. Yellin, R.E. Strom, Protocol specifications and components adaptors, ACM Trans. Program. Lang. Syst. 19 (2) (1997) 292–333.
[50] N. Yoshida, M. Berger, K. Honda, Strong normalisation in the Pi-calculus, Inform. and Comput. 191 (2) (2004) 145–202.